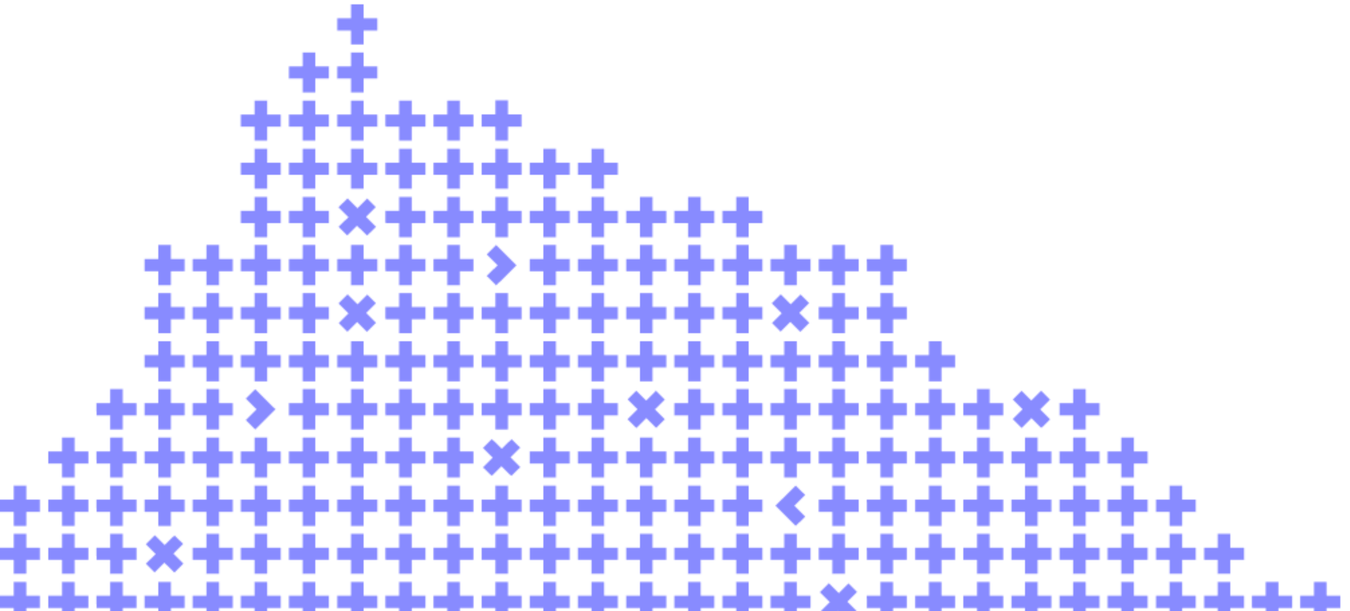
On one interesting generalization of the Leaky Bucket algorithm and Morris's counters

Dmitrii Kamaldinov



High Load Armenia

Co-organizer

Yandex

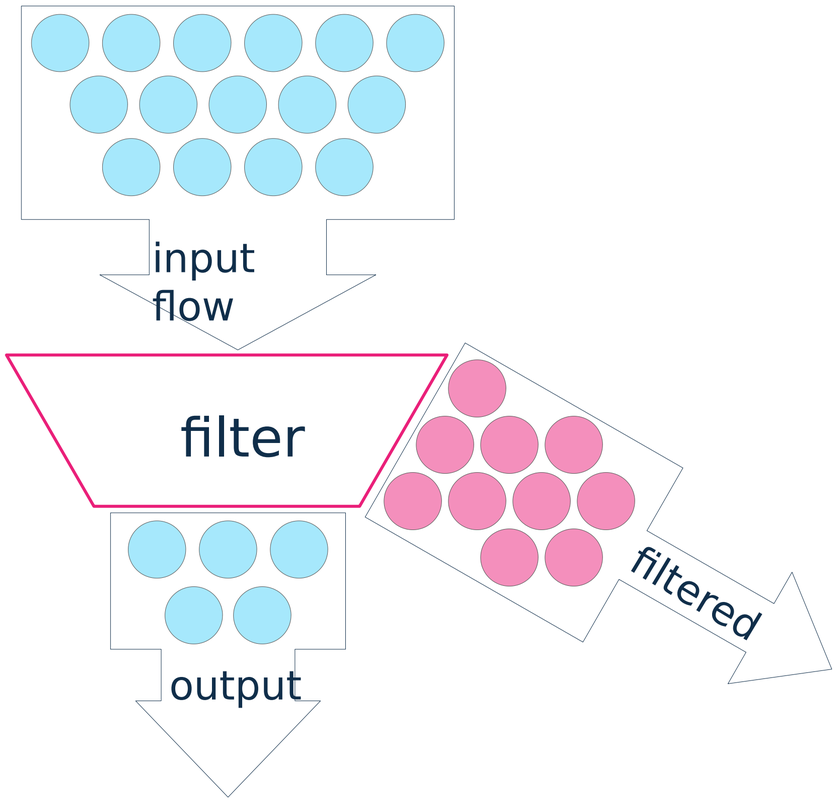
Qrator Labs is a DDoS attack mitigation and BGP security company

Rate Limiting

Rate limiting – preventing the frequency of an event from exceeding some constraint

Examples:

- preventing resource starvation
- load balancing



QRATOR = Queue Rater

Heavy Hitters

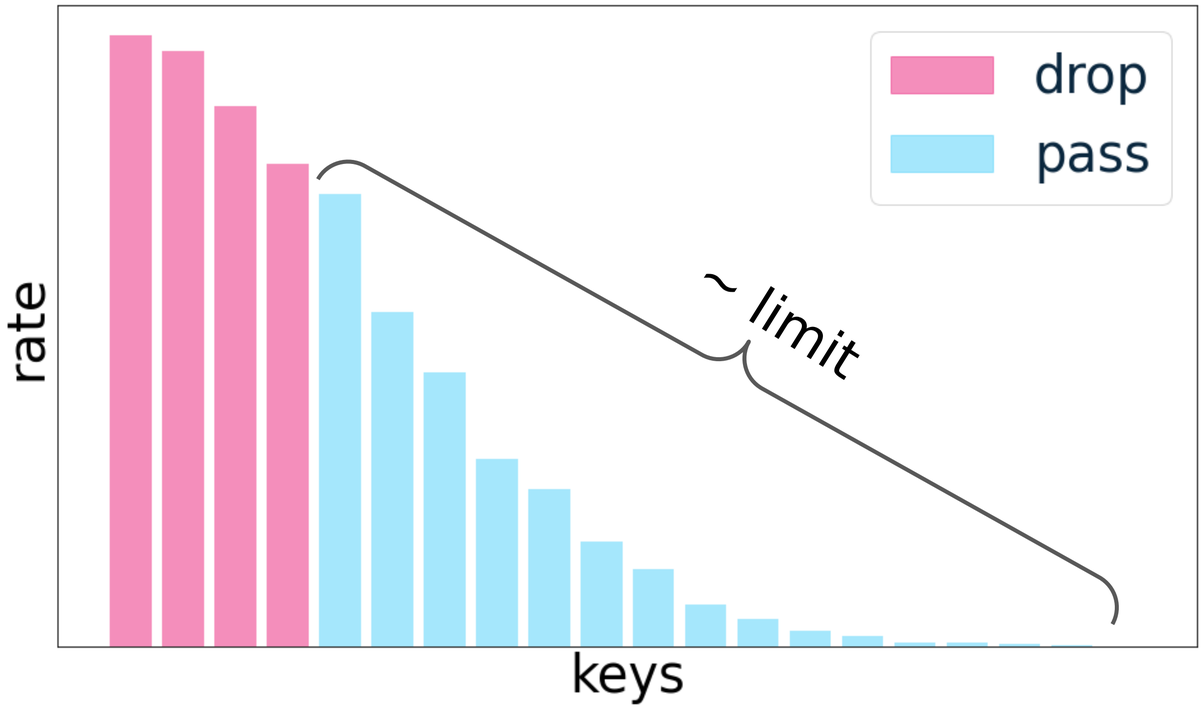
The *Heavy Hitters Problem* refers to detecting the most frequent elements of the stream

Applications:

- traffic analyzing/filtering
  - detecting anomalies (attacks, spam activities)

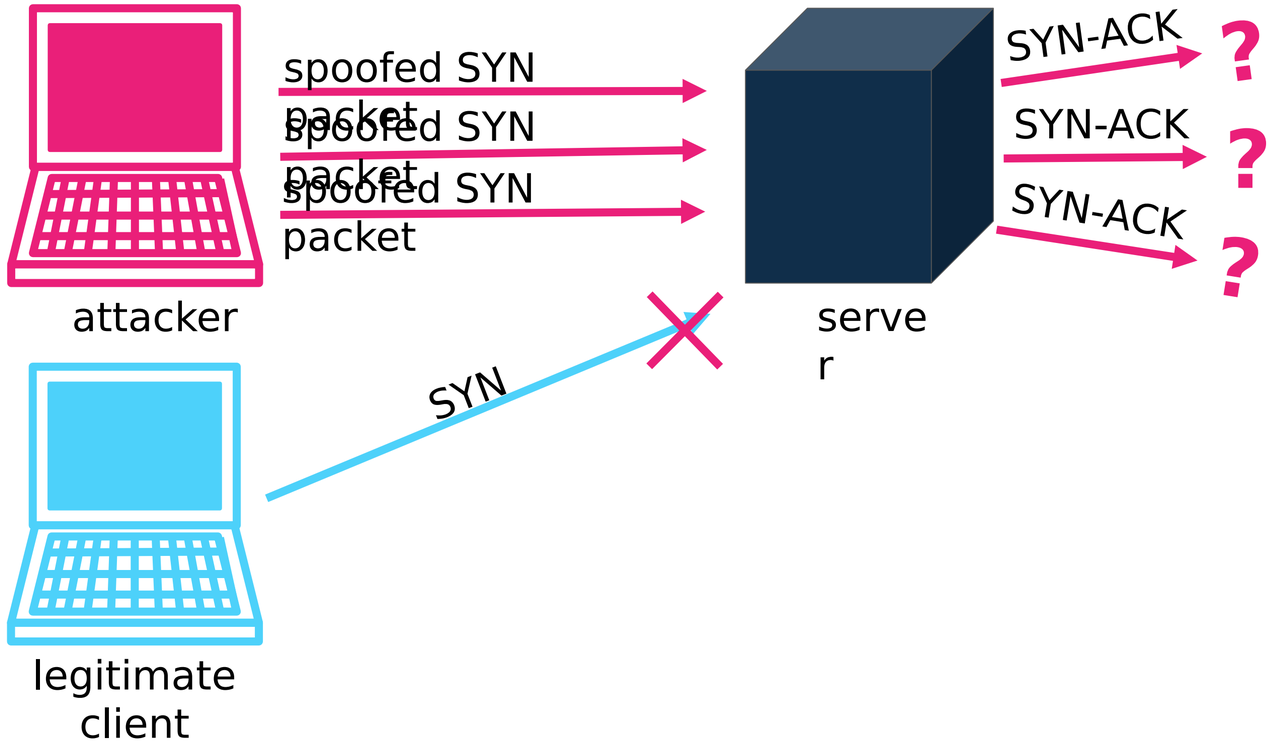
Conditional Rate Limiting

1. limit rate to a given constant
2. drop as few *unique* elements as possible
3. N.B. (2) ⇔
   drop only the most frequent elements
   (heavy hitters)



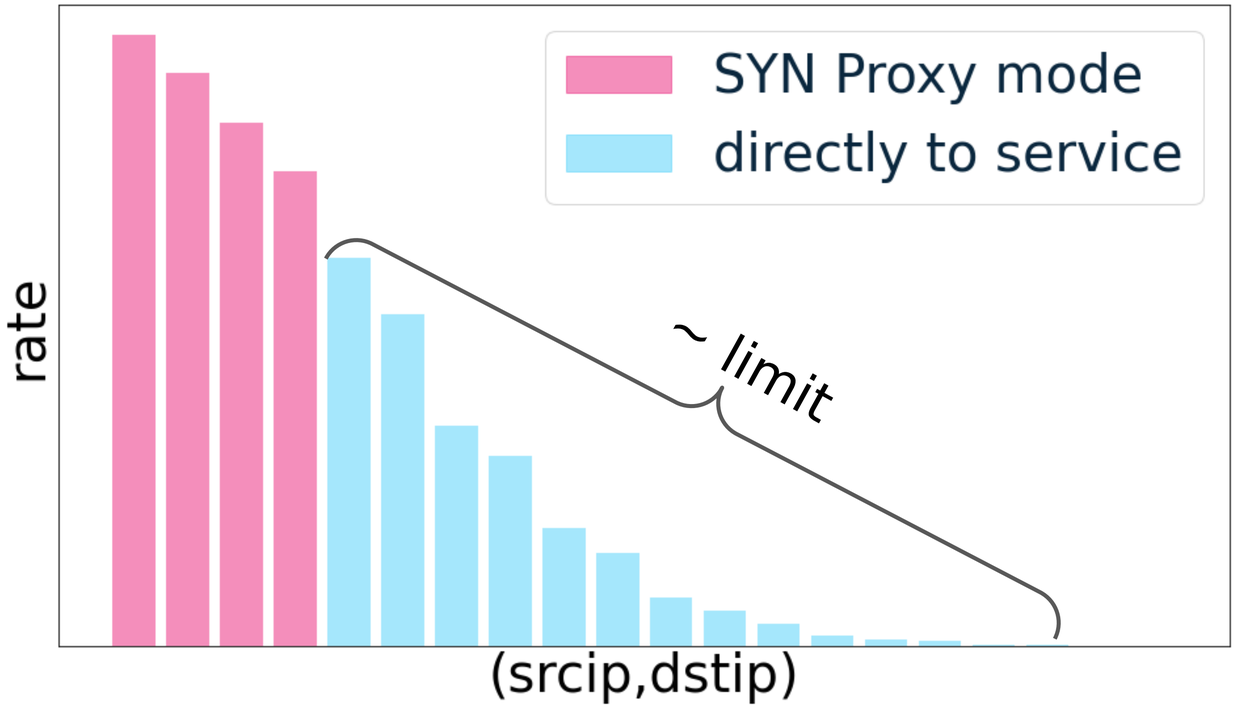
Example 1. SYN Flood.

- attacker initiates a connection (sends a SYN packet)
- but does not finalize it
- the source address can be *spoofed*!
- server has to spend resources waiting for half-opened connections =>
- a legitimate client may be denied a connection



Example 1. SYN Cookies.

- *SYN Cookie* is the most used technique to resist the SYN Flood attack
- Drawbacks:
  - TCP options are discarded
    - e.g. window size!



=> It is reasonable to reply with a SYN Cookie as rarely as possible

- => conditional rate limiting with key (srcip,dstip)

Example 2. Ingress Services.

- when protecting *ingress services* we are able to see only incoming traffic
- we cannot understand which traffic is legitimate
- the only way we can protect such a service is rate limiting

- *internet providers* are the most common example
- an internet provider gives its addresses to its clients
- only some of them might be under attack
- it is reasonable to affect as few of the provider’s addresses (provider’s clients) as possible
- => conditional rate limiting with key dstip

Example 3. MOS vs Packet Loss in Case of VoIP.

Losing even a small part of packets can decrease user satisfaction (Mean Opinion Score) significantly

So, if we are to limit traffic while keeping as many users “happy” as possible, we should try to drop packets belonging to only a small subset of them

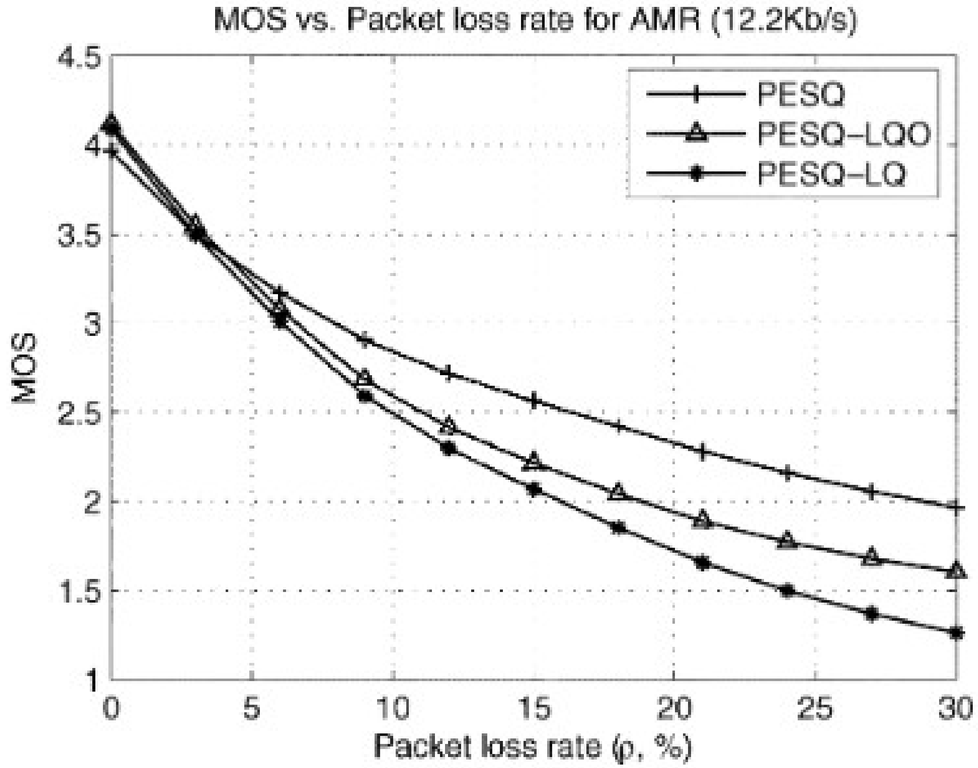


Fig. 6. MOS versus packet loss rate ρ for AMR codec.

Paper: Voice quality prediction models and their application in VoIP networks

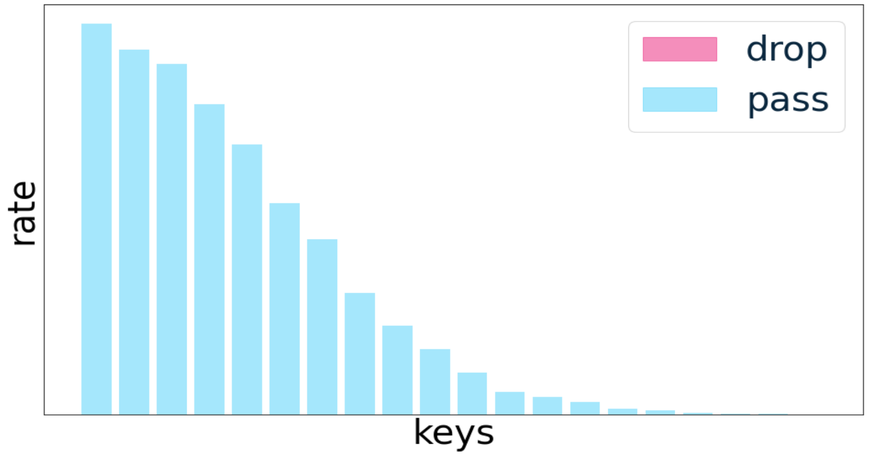
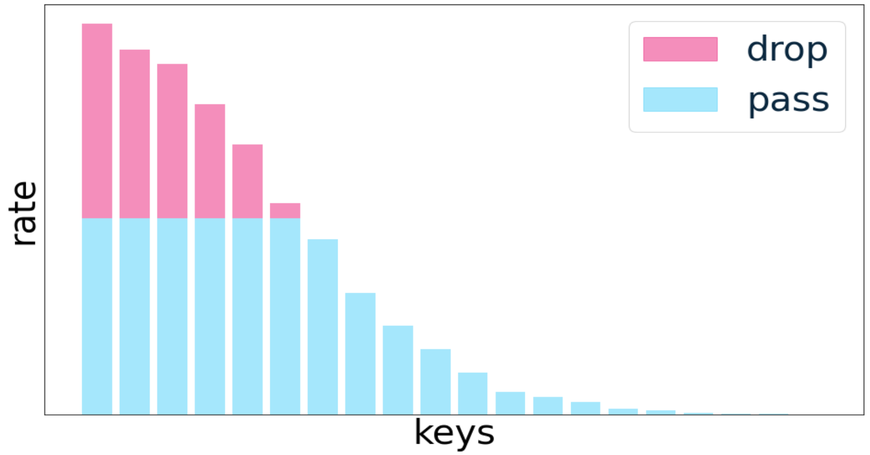
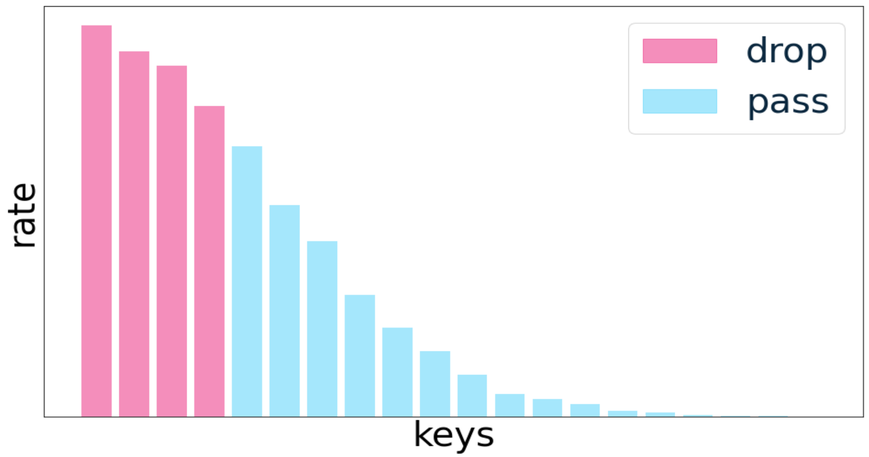
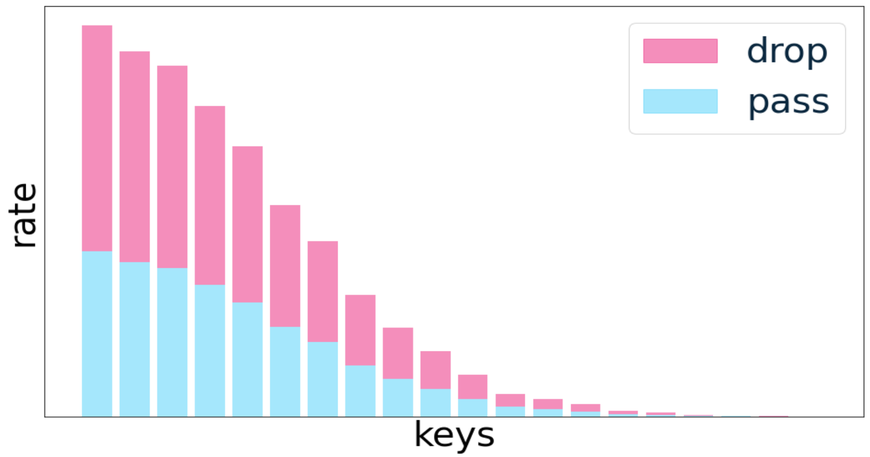
NGINX’s limit_req

```
limit_req_zone $binary_remote_addr zone=perip:10m rate=1r/s;
limit_req_zone $server_name zone=perserver:10m rate=10r/s;
```

Common practice:

- try to limit total service rate by limiting ip rates (zone=perip)
- sometimes this does not help (!), thus total limit is also required (zone=perserver). N.B.
  - when limiting with zone=perserver, keys are not distinguished
  - it’s not worth increasing the number of perip counters (zone size)

Conditional Rate Limiting VS NGINX’s limit_req

	Conditional Rate Limiting	NGINX’s limit_req
total rate does not exceed the limit (legitimate)		
total rate exceeds the limit (attack)		

Leaky Bucket Algorithm

Problem: decrease stream rate down to given **LIMIT**.

How *leaky bucket* works:

- Create a virtual *queue/buffer (bucket)* of accepted stream elements
- Elements from this buffer are removed with a fixed rate of **LIMIT**
- The newly obtained element gets either added to the queue if it is possible (queue is not full)
- Or dropped

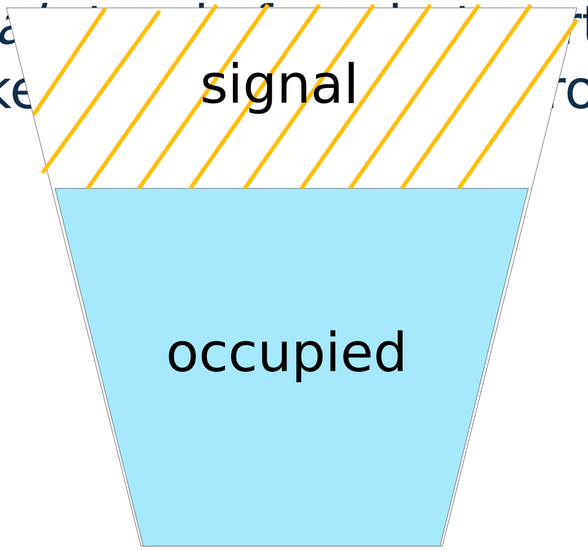
Water leaks out of bucket with a fixed rate

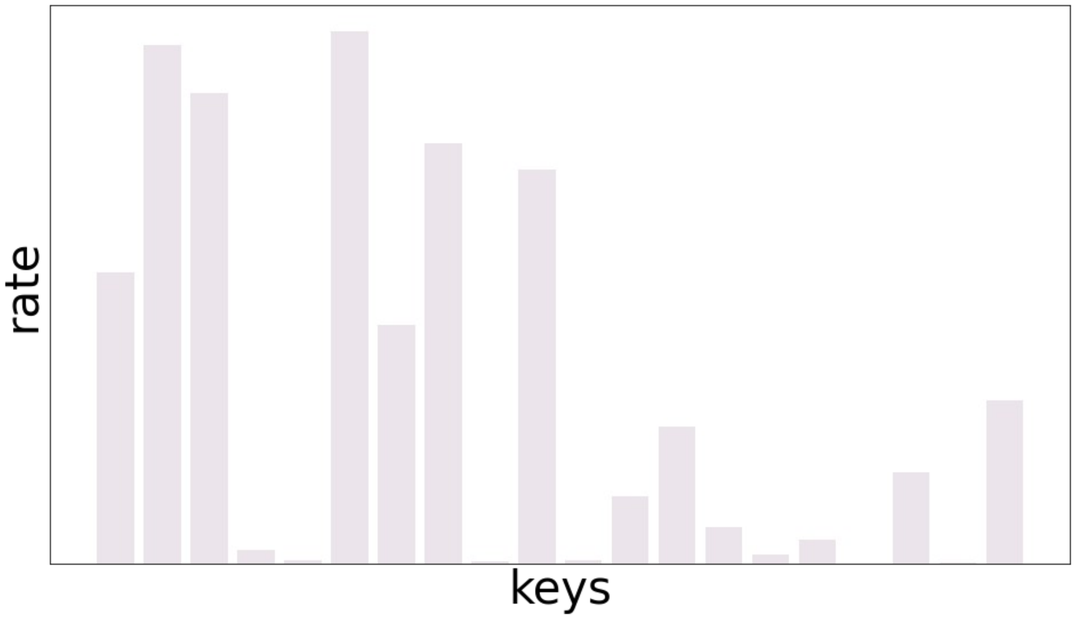
Water spills past the overflowing bucket

Value Filter Algorithm. Structure.

Again, we are to reduce stream rate down to given **LIMIT**

- Create a counter per each event type
  - *rank = (counter / sum of all counters)*
- Create a leaky bucket with respect to **LIMIT**
  - *signa* t of the bucke om 0 to 1)

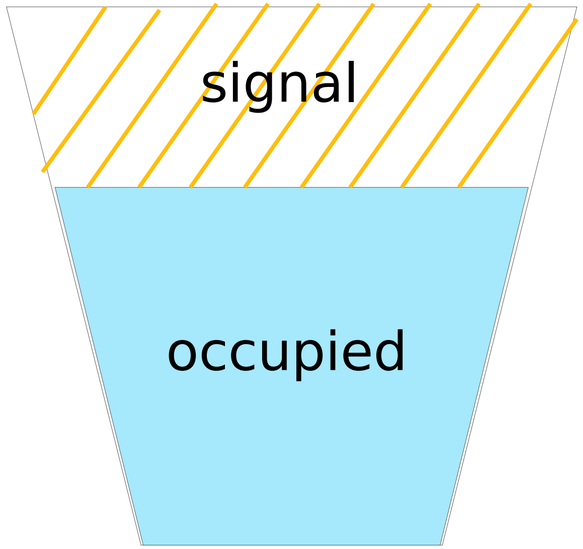


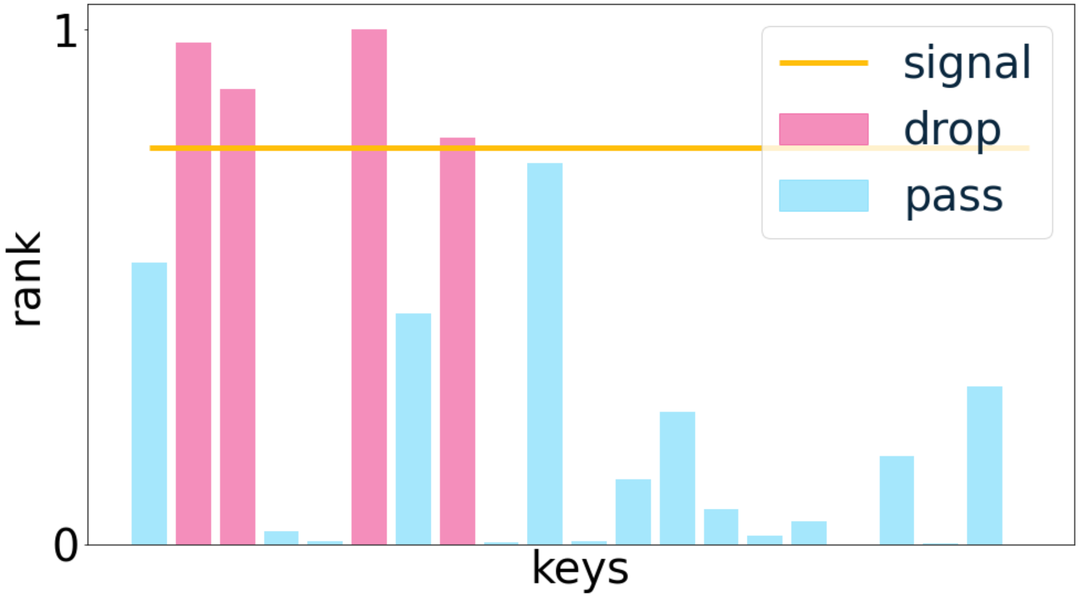


Value Filter Algorithm. How It Works.

When a new event hits

- Increment its counter
- Compare its rank with the signal
  - if less, pass and update leaky bucket

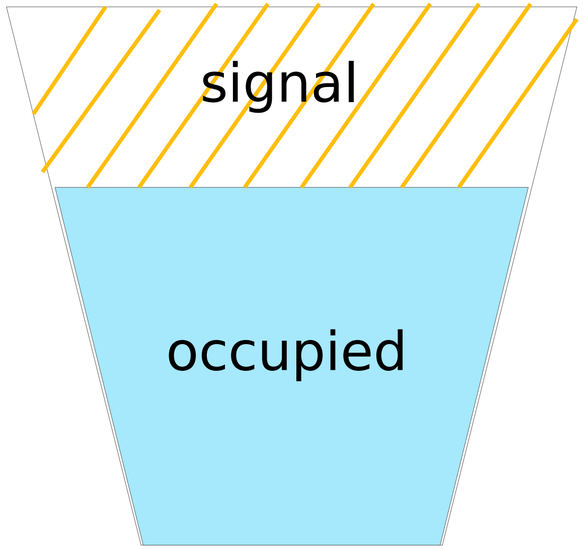


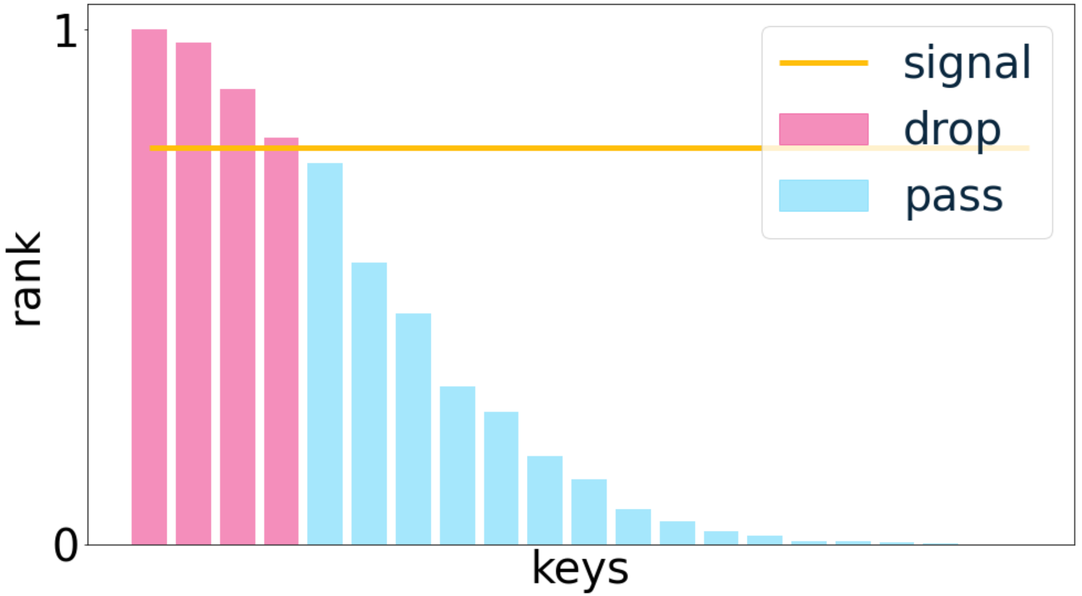


Value Filter Algorithm. How It Works.

When a new event hits

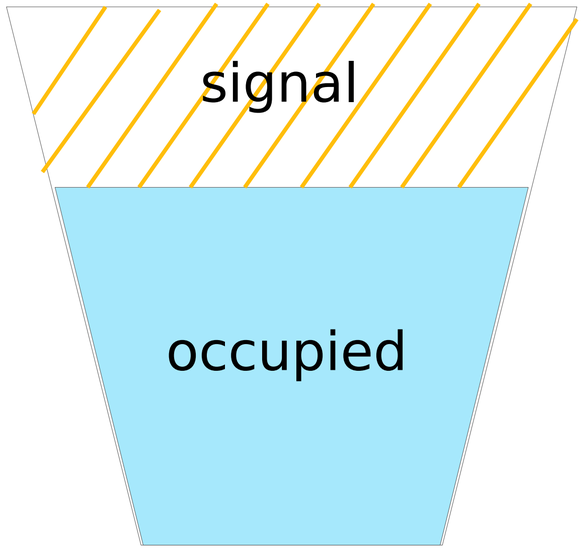
- Increment its counter
- Compare its rank with the signal
  - if less, pass and update leaky bucket

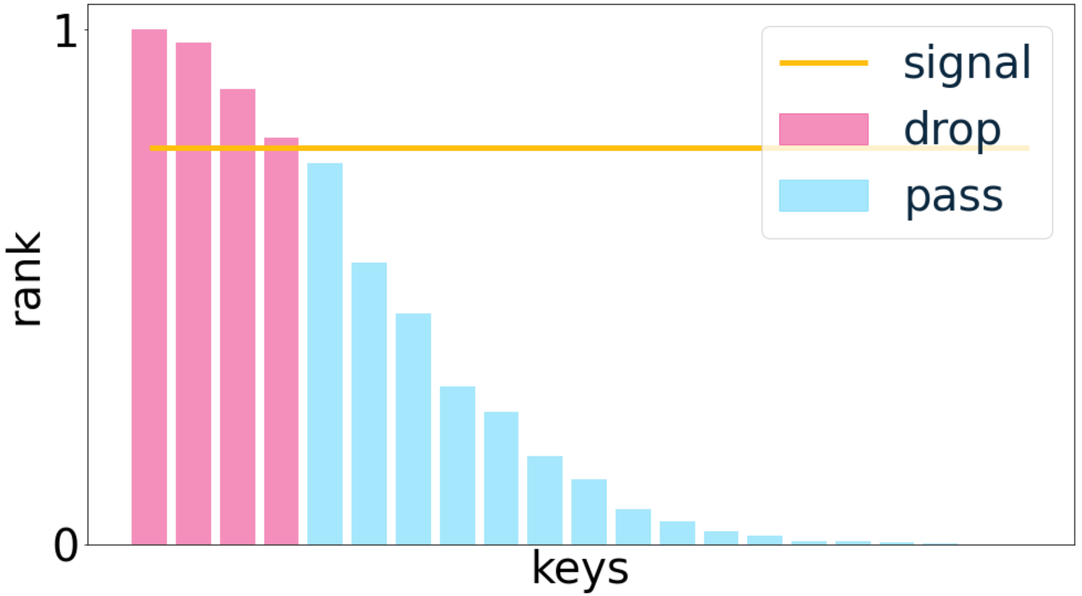




Value Filter Algorithm. Why It Works.

- if at the moment we pass more than **LIMIT**
- the bucket fills up =>
- its occupancy rate grows =>
- the signal decreases
- we pass less





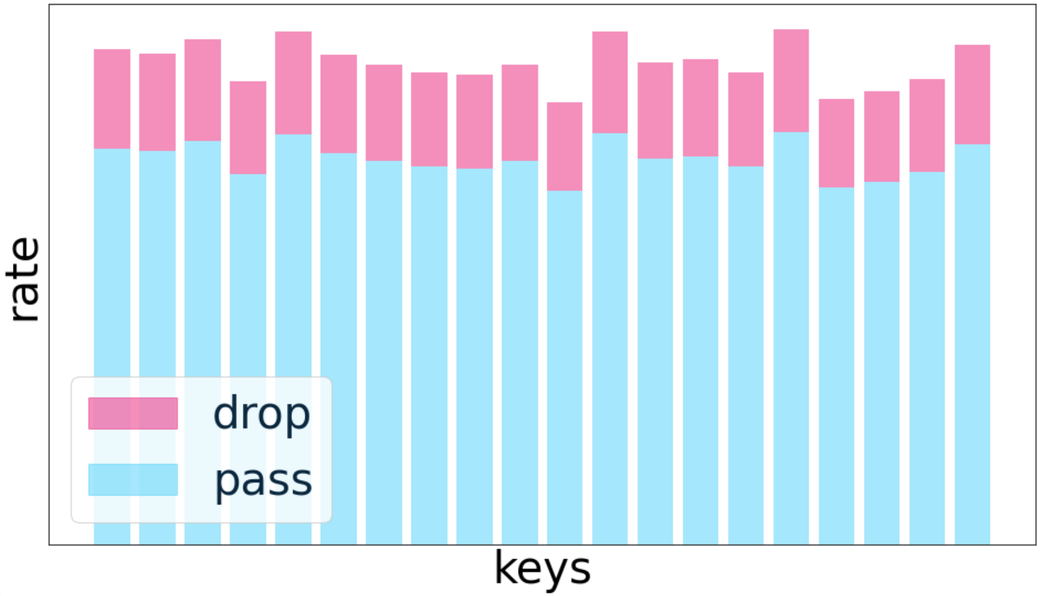
Value Filter VS Leaky Bucket

Leaky Bucket	Value Filter
update complexity is O(1)	update complexity is O(1)
limits total rate to a given constant	limits total rate to a given constant
does not distinguish keys	drops only the most frequent keys when limiting the rate
requires little memory	requires a counter for each key

++ High Load Armenia

Value Filter Algorithm. Drawbacks.

- Uniform rate distribution
  - => there are no most frequent elements
  - how to drop “as few unique as possible”?



Idea: let’s prioritize the keys we’ve been passing the most

Value Filter Algorithm. Drawbacks.

What if there is not enough memory for a counter per each key (IPv6)?

Use one counter for several keys:

- unite keys with the same hash value
- more specific: counters for subnets, not IPs

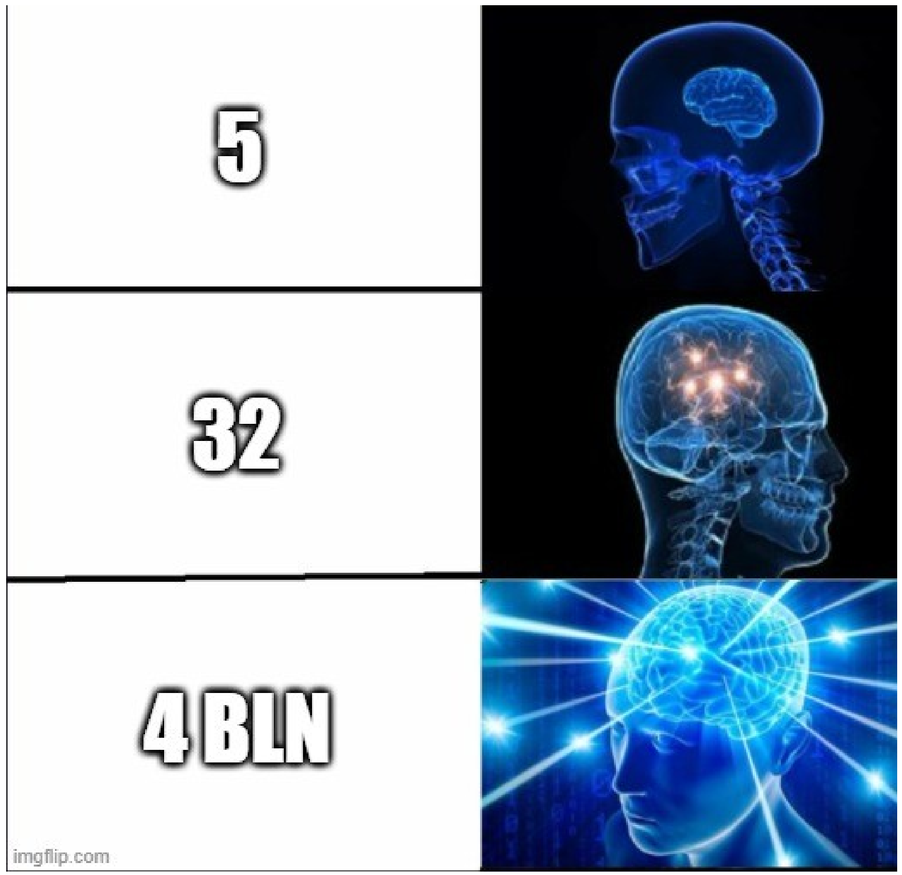
(!) The more precise counters are, the better outcome we get
=> it is reasonable to have as many counters as possible

Morris’s Counters

Counters with value of **X** updates with probability $\frac{1}{2^X}$ of (½)

Counter Value	Update Probability	Average number of events needed for update	Estimated total number of events until now
0	1	1	0
1	1/2	2	1
2	1/4	4	1+2=3
3	1/8	8	1+2+4=7
4	1/16	16	15
5	1/32	32	31

Estimated total number of events for counter value of **X** is $2^0 + 2^1 + \ldots + 2^{X-1} = 2^X - 1$



Counting on the fingers of one hand

Morris’s Counters

```
class Morris:
  def __init__(self):
    self.X = 0

  def _prob(self):
    return pow(2, -self.X)

  def update(self):
    if random.random() < self._prob():
      self.X += 1

  def estimate(self):
    return pow(2, self.X) - 1
```

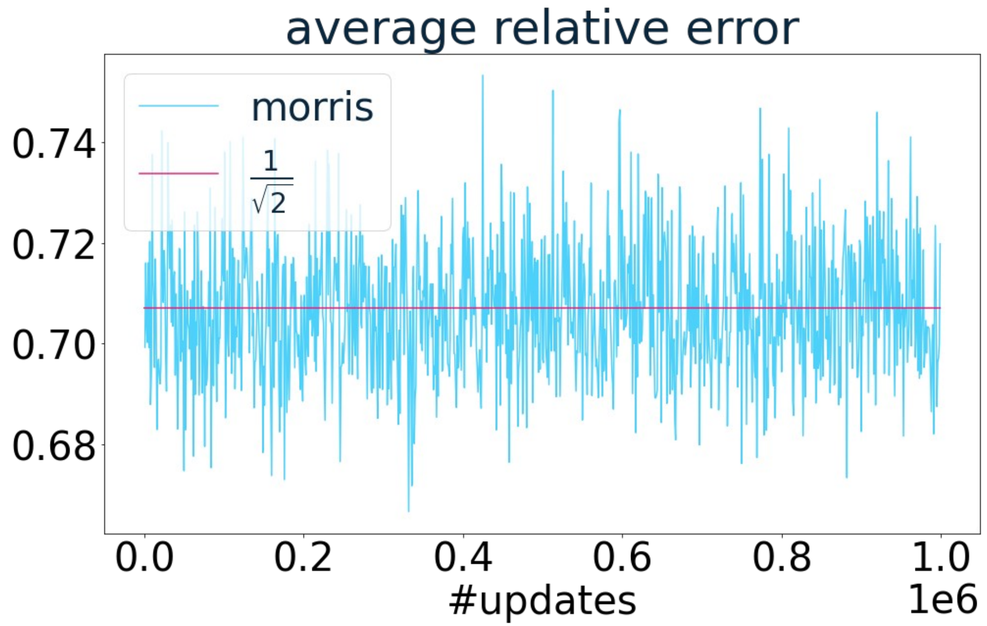
Raw value we store in memory

Update probability depends on current raw value: $prob(X) = 2^{-X}$

Estimated total number of events for raw counter value of X is $2^{X} - 1$

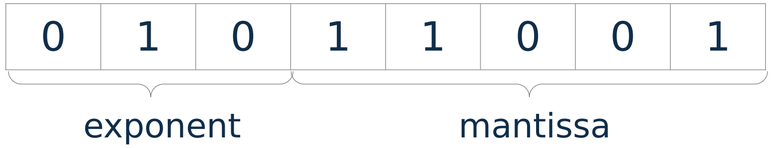
Morris’s Counters. Main Properties.

1. **n**-bit Morris’s counter allows counting up to $2^{n} - 1$
   - or, counting up to **N** requires O(log log **N)** bits
2. estimate is unbiased:
   - after N updates
     avg[est(**X**)] = **N**
3. expected relative error of the estimate is bounded:
   - after N updates
     std[est(**X**)] / **N** ~ 1/sqrt(2)

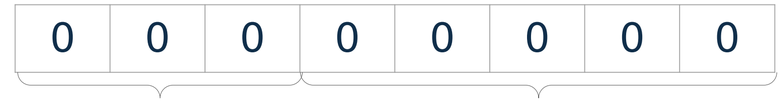


Morris’s Counters. Mantissa+Exponent Extension.

Splitting counters bits into two parts: *exponent*, giving update probability, and *mantissa*

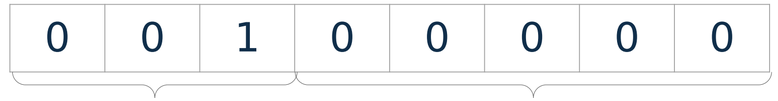


Counter Value	Update Probability	Average number of events needed for update	Estimated total number of events until now
0	1	1	0
1	1	1	1
2	1	1	1+1=2
…	…	…	…
31	1	1	31*1=31
32	1/2	2	32*1=32
…	…	…	…
63	1/2	2	32*1+31*2=94
64	1/4	4	32*1+32*2=96
…	…	…	…
96	1/8	8	32*(1+2+4)=224



here exponent gives update probability of (½)^0 = 1

mantissa changes from 0 to 32



here update probability is 1/2

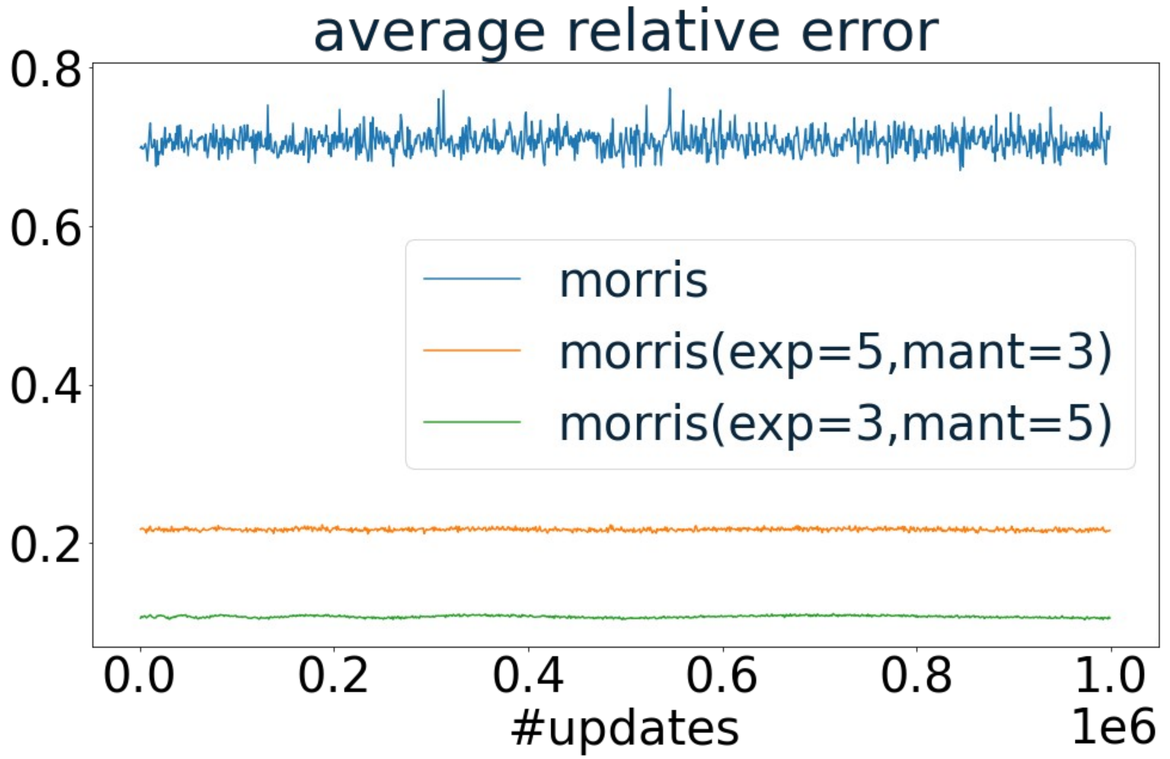
and again we need 32 successful updates for further probability reduction

update probability 1/4

update probability 1/8

Morris’s Counters. Mantissa+Exponent Extension.

Why is this even useful?



- maximal accountable number of events decreases
- but the relative error of the estimate gets better!

Summary

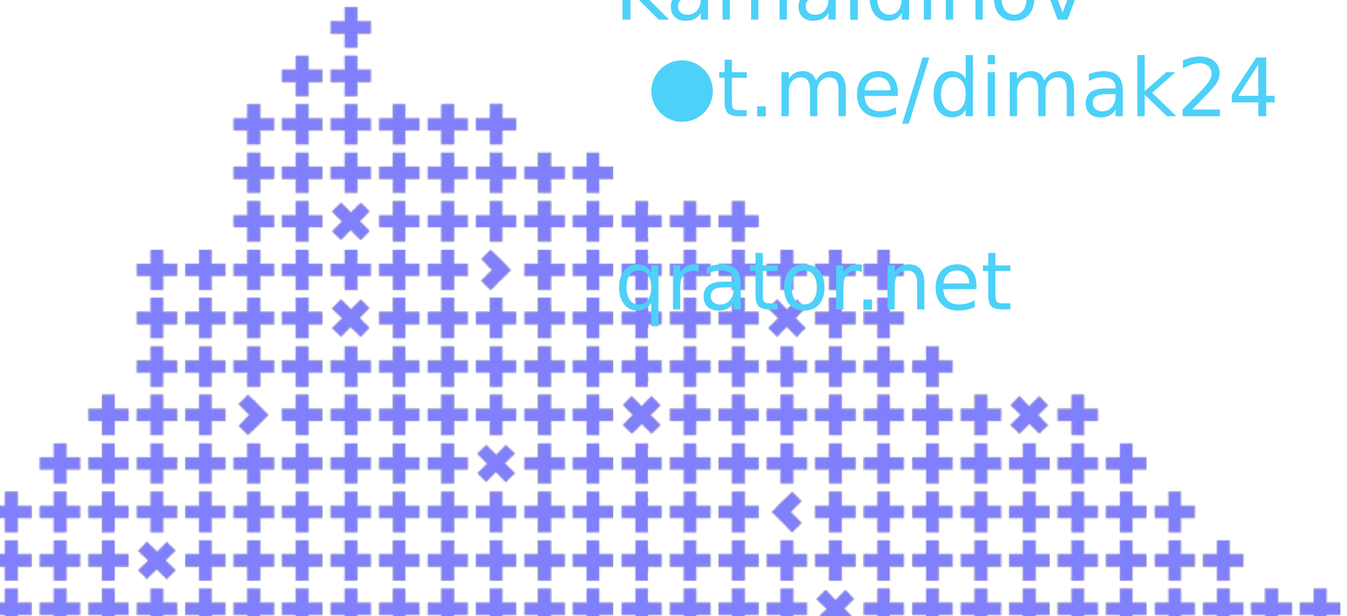
1. Conditional Rate Limiting
2. Value Filter = Leaky Bucket + Counters
   - what are **keys**?
     - dstip
     - (srcip,dstip)
   - **counters**
     - taking into account temporary nature of rates
     - using some tricks to save memory
   - **volume of the bucket** (burst) affects how quickly the algorithm reacts to changes in the flow pattern
3. Morris’s Counters
   - Mantissa+Exponent Extension

Leave your feedback!

You can rate the talk and give feedback on what you've liked or what could be improved

Dmitrii Kamaldinov

t.me/dimak24



qrator.net

Co-organizer

Yandex